游易-程序主题分享合辑

知识管理部 等 2022.06.07 15:37 6242 321 184个资源

汇总从101至今的游易征稿文章合辑。

使用C++11和Boost简化多线程编程

叶天才 2016.02.19 19:51 1634 15 9 查看原文

> C++11, Boost, Lock-Based, Lock-Free, CAS, Memory Barrier，乱序

1. 线程安全的对象生命期管理

C++要求程序员自己管理对象的生命期，这在多线程编程中，变得尤为困难。因为仅仅依靠裸指针，无法确定对象是否还存活，当一个线程试图使用共享对象时，需要考虑多种竞态：

1. 如何保证执行成员函数前，对象还存活着？（没有被其他线程析构？）即使执行成员函数前，对象是存活的，又该如何保证执行过程中，对象不会被其它线程销毁呢？

2. 有人会说，销毁对象后，立马把对象指针置为NULL，这样，其它线程读到这个变量时，先判断是否为NULL，就可以知道对象是否存活啦？类似下面的代码中，使用SAFE_DELETE销毁对象，然后，使用变量前，判断指针是否为NULL：

```
01 int *obj;
02
03 #define SAFE_DELETE(x) do { delete x; x = NULL; } while (0);
04
05 void use_obj()
06 {
07     if (obj) // 此处判断obj != NULL，然后线程被挂起
08     {
09         *obj += 1;  // 线程切回来时，obj可能已经被其它线程销毁
10     }
11 }
```

在单线程中，这是完全可行的，然而，多线程中，由于执行顺序可能被其它线程插入，这种方法无法保证工作。

解决方案：使用智能指针shared_ptr, weak_ptr帮助管理对象的生命期。

拓展阅读：关于这方面的分析，在陈硕的《Linux多线程服务器端编程》的第一章中有十分详细的讲解，包括race_condition的分析，如何正确的使用智能指针进行线程安全的对象生命期管理，本文在此就不再赘述。

2. 基于锁的多线程编程设施（Lock-Based）

在C++11以前，为了进行跨平台的多线程编程，不得不自己封装各个平台的api差异，或者使用一些第三方库。现在C++11已经在标准库中内置了这些多线程编程设施，包括

1. 线程的创建、sleep等操作

std::thread

2. 互斥锁

std::mutex, std::timed_mutex(带有超时的互斥锁), std::lock_guard(方便互斥锁的使用)

3. 条件变量

3. 无锁的多线程编程设施（Lock-Free）

1. 原子类型（Atomic types）

所谓原子类型，即对该类型变量的某些操作（读取、赋值、自增等等）可以是原子的（操作一旦开始，即不会被打断，直到操作结束）

C++11提供了原子类型模板std::atomic<>，使用成员函数is_lock_free可以知道该模板的特化类型，是否是原子的。内置的std::atomic<int>, std::atomic<long>, std::atomic<bool>, std::atomic<float>等类型在大多数主流平台均是LockFree的，即他们的读取赋值(atomic_load, atomic_store)， 自增、自减（atomic_fetch_add, atomic_fetch_sub)等操作，一旦开始，即不会被其它线程打断，直到执行完毕。

这些基本的原子类型，保证了在多线程环境下，对共享变量的一些无锁操作是线程安全的。

2. 内存模型（memory model）

内存模型的提出，主要是针对多处理器下，程序的乱序执行。比如下面的一段代码：

```
a = b;
c = 1;
```

在多处理器下，执行顺序可能发生变化，可能在其他线程下，会看到c=1的操作，先于a=b的操作发生。这种乱序执行可能是由于以下情况：

1. 编译期优化。可能因为某些原因，编译器为了优化性能，将操作互换（只要指令互换不影响单线程下的程序逻辑）

2. 运行期乱序。众所周知，现代cpu是多级缓存的结构，因此，实际cpu指令执行a=b时，需要先在cache中查找b，如果发现cache miss，则会向主存fetch b的值，那这个时候如果cpu核心干等着，岂不是浪费cpu的性能吗？因此，cpu会在得到b的值前，就先执行下面的命令c=1，那么，在其它cpu核心看来，似乎就是c=1发生在a=b之前了。注意，只要不影响程序在单cpu核心上的运行逻辑，这种硬件级的优化就是可能产生的。

我们可以看到，无论是编译期，还是cpu硬件本身，都只保证了程序指令在单线程单核心下的执行结果与顺序执行的结果相同，在这种情况下，程序“看起来”是顺序执行的。所以，在多线程环境下，我们为了保证一些操作共享变量的操作不产生乱序，必须添加一些指令，告诉编译期和cpu，“此处对某种类型的内存读写操作请不要优化”。即添加memory barrier

C++11提供了6种语义的内存 模型，以对内存的操作进行约束：

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

内存模型的本身就是十分复杂的，大多数情况下，我们使用互斥锁等同步原语编程时，并不需要关注内存模型，然而，当编写LockFree的代码时，则必须了解这些模型，以编写出正确高效的LockFree代码。要完全搞懂内存模型，本文的篇幅是完全不够的，故在此不再继续深入下去，有兴趣的同学可以参考《c++ concurrency in action》一书中第五章的阐述，以及《深入理解并行编程》一书（电子版：http://ifeve.com/perfbook/）。

3. CAS（campare and swap）操作

CAS操作的伪代码如下：

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

在大多数硬件上都实现了CAS操作指令，保证了CAS的原子性。众所周知，CAS操作是实现许多LockFree数据结构的重要原语，C++11对此也有了实现，atomic type的成员函数包含了compare_exchange_weak, compare_exchange_strong，可以方便的atomic type的变量进行CAS操作，从而实现一些LockFree的程序。关于LockFree的研究本身是一个庞大的课题，我们在此也不继续叙述，有兴趣的同学可以查找上门提到的两本书中的相关章节进行学习。

需要注意的是，LockFree并不是无锁，只是粒度更小的锁而已，本质上LockFree只是通过CAS操作进行不断的“失败-尝试”的过程，而LockFree程序相比Lock based的程序更加难以debug，出了问题时，十分难于排错，因此，在对实时性要求不高的程序中，并不推荐使用Lock Free的程序，比如游戏服务端程序。仅仅在实时性要求高的程序中，LockFree才有价值，比如游戏客户端的实时渲染。如果非要使用，推荐使用一些已经确定无误的Lock Free的数据结构，例如boost中的LockFree模块中的队列、stack等。在实际编程中，常见的一些操作，例如任务分发、日志，使用LockFree的queue已经完全足以满足需求。

4. Boost中一些好用的轮子

Boost的代码质量还是很高的，在大多数情况下，boost中已经实现的功能，我们还是没有必要再造轮子。关于并发编程，boost中有很多基础轮子可供使用：

1. Asio库：Proactor的网络库

2. Coroutine库：C++的协程库（切换用户态线程上下文，想想都感觉屌屌的！）

3. LockFree库：提供了lock free的queue，stack

总结：目前C++11和Boost中的一些常见的多线程编程设施基本完善，大家可以更轻松的在多线程的世界遨游了！

全部评论 9

请输入评论内容

还可以输入 500 个字

(可添加1个视频+5张图片) 匿名 评 论

最热 最新

马文涛

9楼 因此，在对实时性要求不高的程序中，并不推荐使用Lock Free的程序

自旋锁和混合锁的第一阶段都是使用CAS重试的。此类锁在临界区很短的场景里都能提高性能吧。

2017-10-24 10:47 回复 0

吴俊琦

8楼 👍

2016-04-28 14:28 回复 0

匿名

7楼 马克

2016-03-25 17:12 回复 0

匿名

6楼

2016-03-21 13:04 回复 0

匿名
4楼 ☺
2016-03-19 13:30 回复 0

匿名
3楼 路过。
2016-03-17 09:30 回复 0

匿名
2楼 赞👍
2016-03-10 15:27 回复 0

匿名
1楼 赞哦
2016-03-08 12:43 回复 0

加载完毕,没有更多了